TDB-ACC-NO: NN9512299

DISCLOSURE TITLE: Handling Shared Objects in Multi-Process Systems

PUBLICATION-DATA: IBM Technical Disclosure Bulletin, December 1995, US

VOLUME NUMBER: 38

ISSUE NUMBER: 12

PAGE NUMBER: 299 - 300

PUBLICATION-DATE: December 1, 1995 (19951201)

CROSS REFERENCE: 0018-8689-38-12-299

DISCLOSURE TEXT:

In a multi-process system such as OS/2*, objects conventionally belong to processes. Objects belonging to the same process can communicate simply by sending messages, but objects belonging to different processes cannot send messages directly to one another. Instead, they must use some other means of inter-process communication, which introduces complexity for the programmer and overheads due to process switching.

The disclosed solution introduces two kinds of objects for use by multiple processes: **shared objects** without process affinity, and **shared objects** with process affinity.

Shared objects without process affinity can be used from any process without a process switch. Their state resides in shared memory, which is accessible and updatable from all processes. Their operations (typically accessing and updating **shared memory**) are independent of the process on which their methods execute. These objects provide concurrency control to ensure serialised access to their **shared memory** state when concurrently executed from multiple processes (or, in multithreaded systems, from multiple threads within a process). When invoking a method of such an object, there is no need to switch processes.

Shared objects with process affinity must execute on a

designated process. However, they can be used from any process, with any required process switching being performed transparently by the object as part of method invocation.

The user of the object (message sender) need not be aware that the receiver object has process affinity and is performing a process switch as part of method invocation. No process switch is performed when the sender of a message is already running on the receiver's designated process.

In all cases, the message sender need not be aware of the characteristics of the message receiver object (unshared of shared, process affinity), since it is the receiver object's responsibility to switch processes only when necessary. This simplifies the design and programming of the message sender, and ensures a minimum of process-switching overhead.

In one example of implementations by Object REXX for OS/2, **shared objects** without process affinity reside in named **shared memory** and are accessible from any OS/2 process, with Object REXX providing the necessary concurrency control. **Shared objects** with process affinity are supported using proxy objects, which are **shared objects** without process affinity and so can be accessed from any process. The proxy object represents the **shared object** with process affinity (its target), and performs the following processing when it receives a message intended for its target:

1. Compare the current process with the target affinity process (which was recorded in the proxy when the proxy was created).
2. If the processes are different, perform a process switch to the target affinity process (using a special server object that runs within the target affinity process).
3. Forward the message onto the target object (now running on its affinity process).
4. If the target method returns normally and a process switch

took
place, the target affinity process server switches back to the
original sending process and returns any method result to it.

5. If the target method raises a terminating condition and a process switch took place, the target affinity process server traps the condition, switches back to the original sending process, and passes an equivalent terminating condition to the sender.

Further refinements of this solution are possible, for example:

For **shared objects** with process affinity, references to the object that are passed to other processes could be transformed automatically to references to its proxy. This would eliminate the possibility of direct references to the object (not its proxy) being passed to other processes, which could cause methods of the object to be run on the wrong process.

For shared objects with process affinity, the proxy object could be eliminated and its logic incorporated into message processing for the target object. This would eliminate the problems that can arise from having two distinct physical objects (proxy and target) for the same logical object, with the possibility of references to the target object being passed to other processes instead of references to the proxy (see previous item).

Process affinity could be applied on a per-method basis instead of a per-object basis. This would allow some methods of an object to cause a process switch whilst others do not.

For example, methods that update a shared object's state could run on a special process with read-write access to the object's memory, whereas methods that only read the object's state could run on any process that has read-only access to the object's memory.

* Trademark of IBM Corp.